Simplifying Analyses with Advanced C++

Christopher Jones
Cornell University

Introduction

Physicists want to spend their time studying the data instead of learning about, writing and debugging code.

Use of advanced C++ coding techniques can help achieve this.

Overview

CLEO

Software Principles

Data Access
- templates
- exceptions

Combinatorics
- operator overloading
- expression templates

CLEO: History

Based at Cornell University

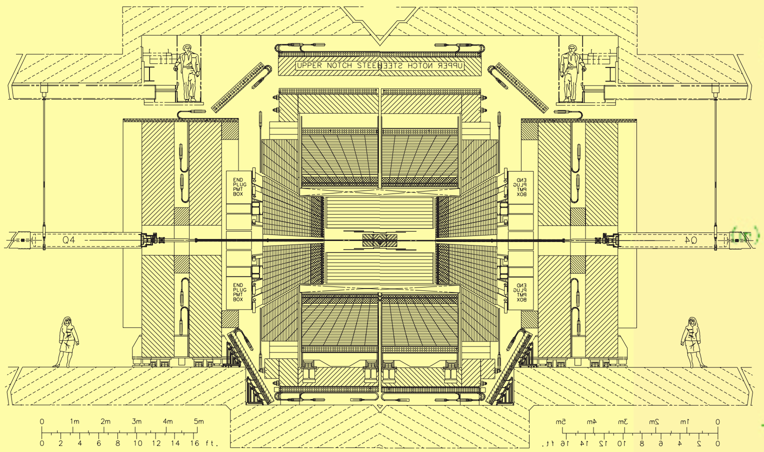
Using the Cornell Electron Storage Ring (CESR)

e+ e- machine with center of mass energy 3-10 GeV

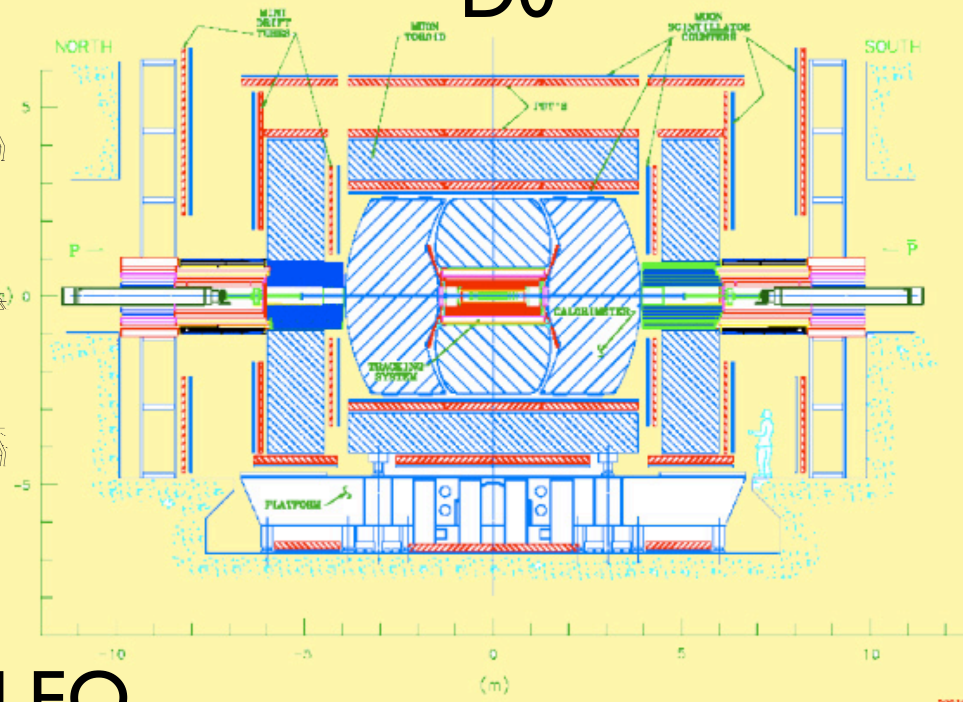
Date	Detector	Studying	Energy
1979	CLEO I	Υ ($b\bar{b}$) resonances	10 GeV
1988	CLEO II	Υ(4s) decays to $B\bar{B}$	10 GeV
2000	CLEO III	Υ(4s) decays to $B\bar{B}$	10 GeV
2003	CLEOc	ψ ($c\bar{c}$) resonances	3 - 4 GeV

CLEO: Detector

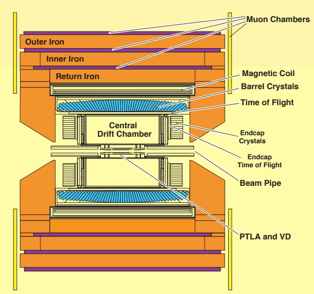
CDF



D0



CLEO



Fermilab Computing
2004/ 12/ 14

CORNELL UNIVERSITY LEPP
LABORATORY FOR ELEMENTARY-PARTICLE PHYSICS

CLEO: Data

Now

nearly 1B events taken

20 M $B\bar{B}$ pairs

3.4 M ψ resonance decays

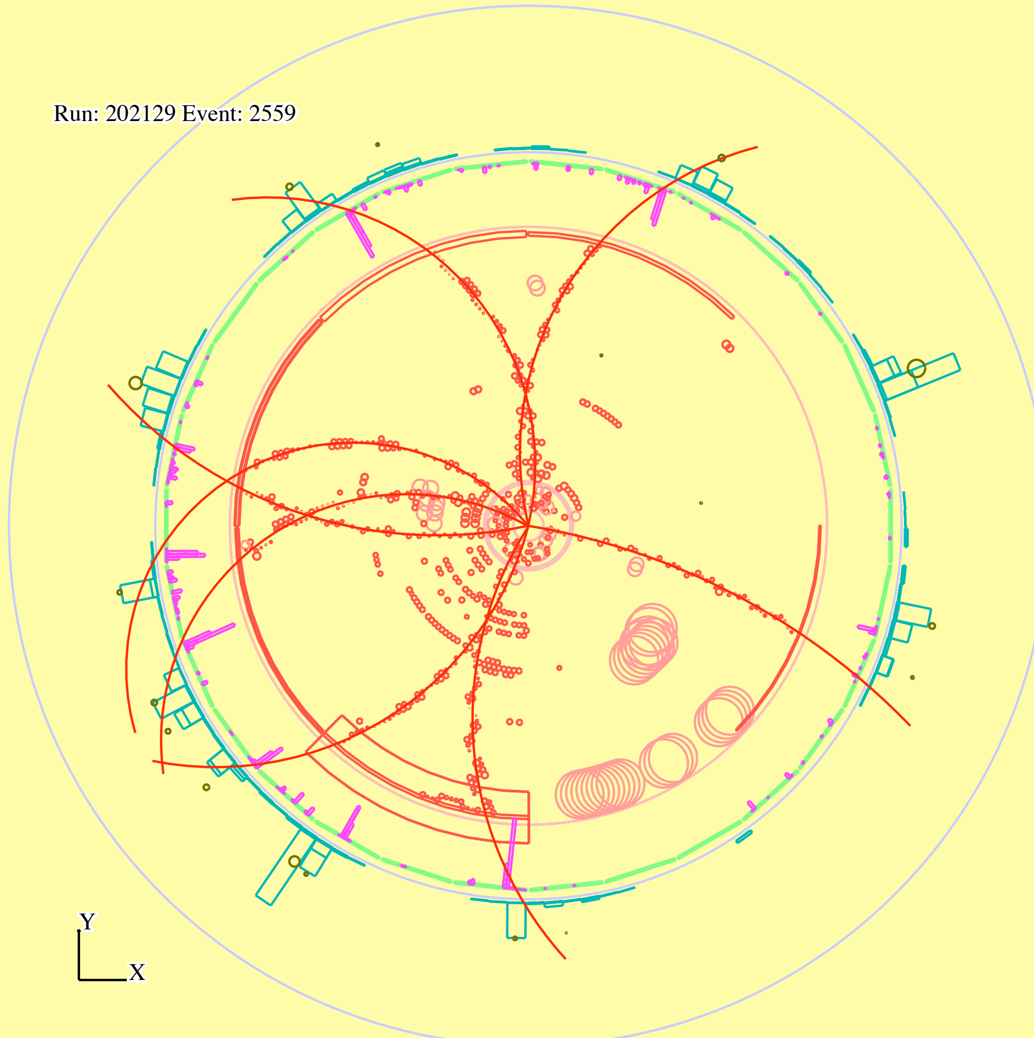
near 100 TB data stored

Future

1B J/ψ events

10s M signal events / analysis

precision measurements
for comparisons with
Lattice Gauge calculations



Fermilab Computing
2004/ 12/ 14

CORNELL UNIVERSITY LEPP
LABORATORY FOR ELEMENTARY-PARTICLE PHYSICS

Software History

CLEO II used a FORTRAN system

96 Summer started work on C++ analysis environment
3 full time postdocs

97 Fall adopted as official CLEO III data access framework

98 Sept had workshop and release

99 Nov used for processing engineering data

00 Oct first reconstruction
14.5 FTE of manpower

Guiding Principles

Physicists want to do physics not program

Concentrate on how physicists think about and use data

Design to be as general purpose as possible
users only have to learn one thing and then apply it everywhere

Impossible to get incorrect data

Make the compiler do the work
keep user interfaces type safe

Make the program do the work
have the program do the bookkeeping, not the user

If it is hard to use it is our fault and we need to fix it

Suez Framework

One C++ Framework for all data access

- Level 3 trigger, online monitoring, calibration, reconstruction, event display, analysis
- only have to learn one system

Dynamic loading of components

- dramatically decreases link time

Tcl command interface

- easy to learn
- third party documentation available

Use multiple sources simultaneously

- can use a previously made skim to drive system to only read events of interest

Data on demand

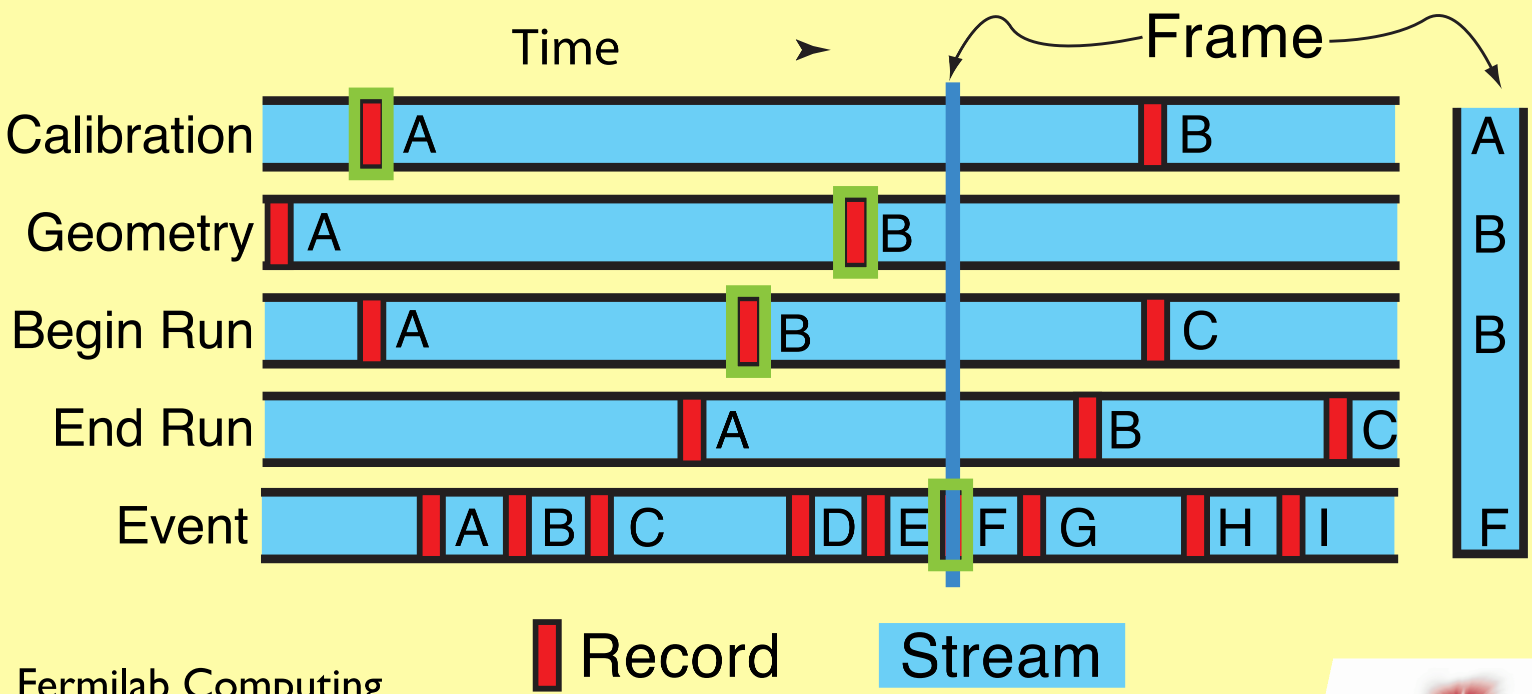
- substantially easier job configuration

Data Model

Use mental model from DAQ

All data is accessed through the **Frame**

Frame: A "snapshot" of CLEO at an instant in time, formed by the most recent Record in each Stream



Data Access

All data accessed using the same syntax

```
Result MyProc::event(Frame& iFrame ) {
   Table<Track> tracks;
   extract(iFrame.record(kEvent), tracks);

   Table<Shower> myPhotons;
   extract(iFrame.record(kEvent), "MyPhotons", myPhotons);

   Item<DBRunHeader> runHeader;
   extract(iFrame.record(kRun), runHeader);
```

Frame holds Records, Records hold data

Type-safe

String only used if do not want default data for type

Table<> and **Item**<> are handles to data

Exception thrown if access fails

Record Design

- Based on a design from Babar
- Type-safe heterogeneous container
- Key-based object retrieval
- Key has three parts
  - Type
    - translated into an integral value at run time
  - two strings (Usage and Production)
    - default object obtained using empty strings
- Object insertion builds Key
- Object held as a void*
  - only private interfaces can see the void* all other interfaces are type-safe
- Object retrieval
  - builds key based on type of variable and optional strings
  - gets object from internal structure
  - casts object to proper type and assigned to input variable

Record's Key

Key built by templates
First time a Key is requested for a type, it is assigned a value

```
template<class T,class Key,class Tag> class HCMethods {

   Key makeKey(const Tag& iTag) {
     static TypeTag<Key> sType = TypeTagTmp<T,Key>();
     return Key(sType, iTag);
   }

template<class T,class Key>

class TypeTagTmp : public TypeTag<Key> {

   TypeTagTmp() : TypeTag<Key>( getValue() ) {}

   static unsigned long getValue() {
     static unsigned long v = TypeTag<Key>::getNext();
     return v ;
   }
```

Specialized Container

Table<T> holds a const PtrTable<T>*

both conform to random access container semantics

- size(), begin() and end(), operator[](), find(), empty(), front(), back()
- Table<> just forwards all calls to PtrTable<>*

PtrTable<T> requirements

items in a list must return a unique value from 'identifier()' method

- Two users talking about track '3' are guaranteed to refer to same object

operator[] finds object via identifier()

- objects internally sorted for fast look up

internally holds T*

- multiple lists can share same objects

externally looks like container of T's

- avoids having to do double dereference of iterators

Template Optimization

PtrTable<T> is used mostly for reading

Specialized for different types of **T::Identifier**

- integral types: use std::vector internally
- other types: use std::map internally

Allows optimal `find(T::Identifier)` method

Allows optimal iterator type

Data on Demand

Designed for analysis batch processing

not all objects need to be created each event

Processing is broken into different types of modules

Providers

- **Source**: reads data from a persistent store
- **Producer**: creates data on demand

Requestors

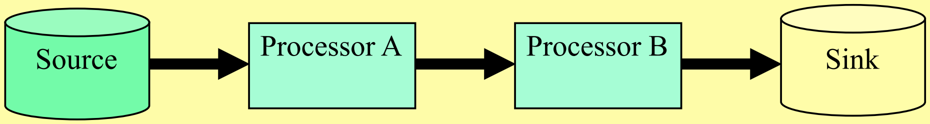
- **Sink**: writes data to a persistent store
- **Processor**: analyzes and filters 'events'

Data providers register what data they can provide

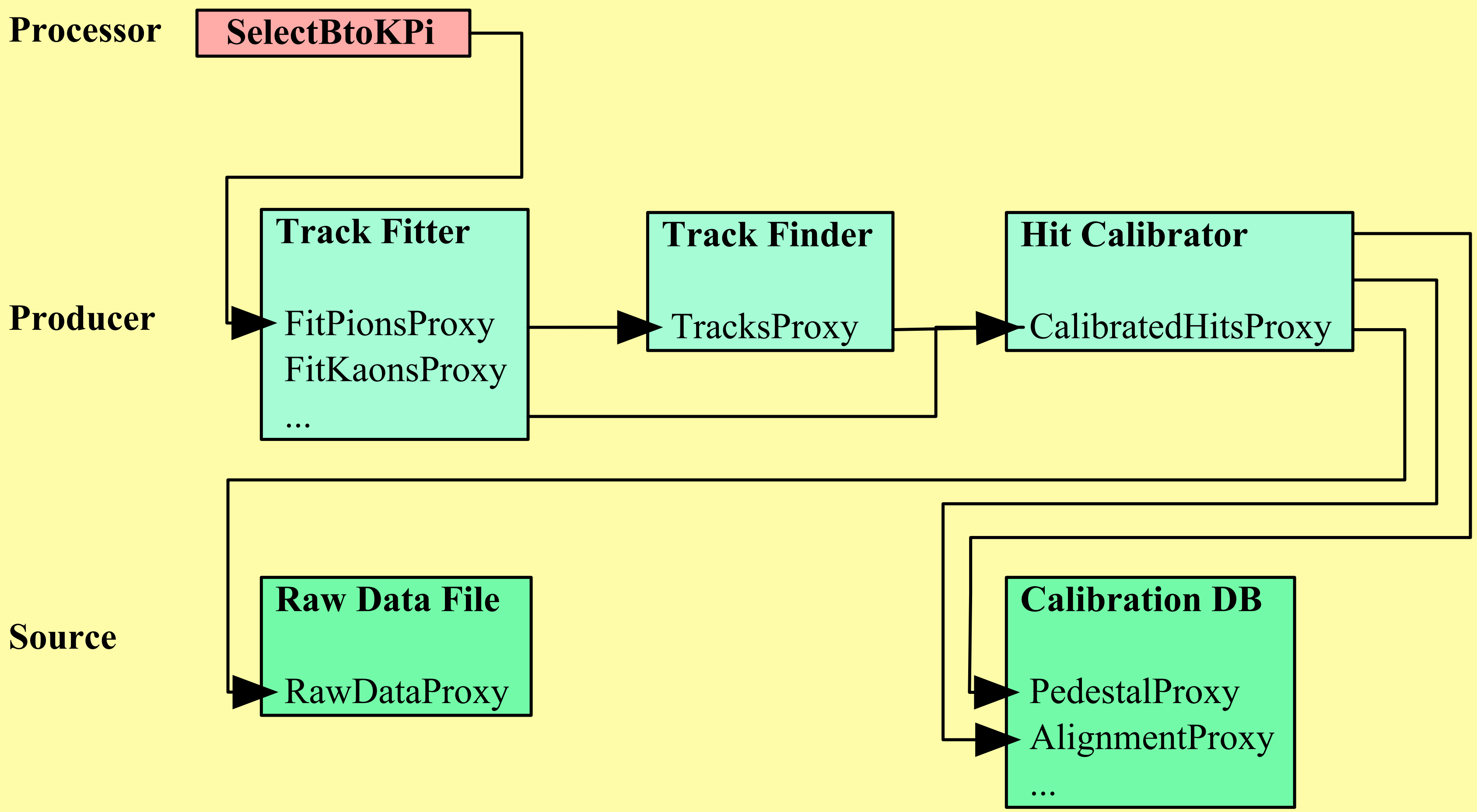
Processing sequence is set by the order of data requests

Only Processors can halt the processing of an 'event'

Physicists only explicitly set order of Processors



Example: Get Tracks



C++ Exceptions

Without an explicitly set processing sequence, isolating the cause of a failure can be tricky

Use of C++ exceptions an absolute necessity

Physicist's code can just assume no problems can occur

- on error, the routine will be aborted
- no messy status checking necessary

Exception safety

- analysis code just uses objects on the stack
- reconstruction code makes use of 'std::auto_ptr' and a custom list holder smart pointer

NOTE: Exceptions added 6 months after start of first reconstruction.
Only took 3 days.

Lesson: Never too late to add exceptions

Exception Traces

Unlike Java, C++ provides no stack trace for an exception

We trace data access calls

- each ‘extract’ call pushes Key onto trace stack
- exiting ‘extract’ call causes index of stack to be moved down one (key remains in list)
- constructor of exception holds index to present top of stack
- caught exception can print data access stack from its original top to the new top

Adds <10% overhead to very fast access calls

```
>> Mon Dec  6 13:16:37 2004 Run:  114277 Event:    7799 Stop: event <<
%% ERROR-JobControl.ProcessingPaths: Starting from GamGamKsKsReadFullProc we called extract for
[1] type "FATable<NavShower>" usage "SplitoffApproved" production ""
[2] type "FATable<SplitoffInfo>" usage "" production ""
[3] type "FATable<NavTrack>" usage "Muons" production ""
[4] type "FATable<NavTrack>" usage "Electrons" production ""
[5] type "FATable<DedxInfo>" usage "" production ""
[6] type "FATable<DedxInfo>" usage "MC" production "" <== exception occured
caught a DAException:
"No data of type "FATable<DedxInfo>" "MC" "" in Record event
This data type "FATable<DedxInfo>" exists, but has different tags.
usage "" production ""
Please check your code and/or scripts for correct usage/production tag."
```

Combinatorics: DChain

Particle combinatorics is tedious and error prone
- must write loops within loops
- must avoid double counting particles
- must avoid double counting because of conjugation

DChain is a package for building lists of decay chains
- decay lists are built by 'multiplying' lists of particles
- understands conjugation
- uses selection functions and objects to decide what decays go into a list
- template based to be experiment independent

```
ChargedPionList pions;
ChargedKaonList kaons;
pions = tracks;
kaons = tracks;
DecayList d0List, dPlusList;
d0List    = kaons.minus() * pions.plus();
dPlusList = kaons.minus() * pions.plus() * pions.minus();
```

Delayed Evaluation

The math expression, **kaons.minus() * pions.plus(),** is not evaluated immediately, instead it produces a **CombinatoricList**

CombinatoricList holds the lists of particles and conjugations

DecayList::operator= does the actual work

- checks if any lists are duplicates and optimizes loops accordingly
- checks if any lists are conjugates of each other
- loops over particle lists keeping only those decays
  - particles do not come from a common 'observable' (e.g. same track)
  - pass user's selection criteria

Selection

Simple function

```
bool myD0s(Decay& );
```

simplest idea to understand
does not work when selection requires info not available from Decay (say the beam energy)
selection function code can not be nested in event processing function

Selection object

```
class MyD0Select : public SelectionFunction<Decay> {
    public:  bool operator()(Decay&);
};
```

member data can hold additional selection information
selection class must be declared external to event processing function

Functional expression

```
SimpleSelector<Decay> d0Sel = abs(vMass-kD0Mass) < 100*k_MeV
                      && abs(vEnergy - beamEnergy) < 100*k_MeV;
```

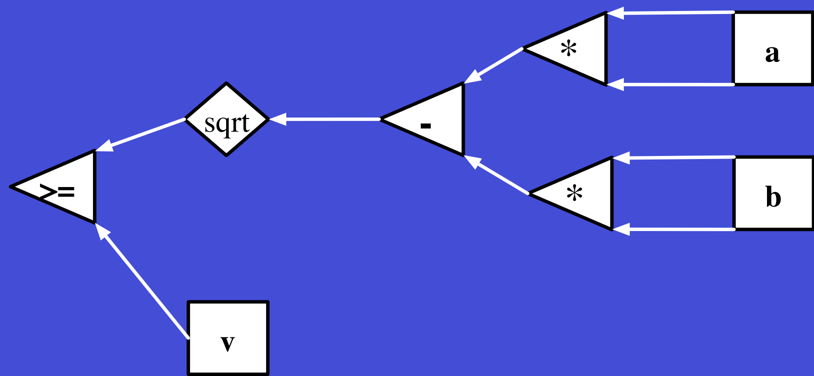
uses expression templates to build a selection object at compile time
external data (e.g. beamEnergy) become member data
expression can be declared next to code that uses the selector
only mathematical and boolean operations can be used for selection (e.g., no loops)

CORNELL UNIVERSITY LEPP
LABORATORY FOR ELEMENTARY-PARTICLE PHYSICS

Expression Templates

Expression `sqrt( a*a – b*b ) >= v`

As a Graph



As a Class

encode expression in template structure

```
GtEqOp<
       SqrtOp< SubOp<
                     MultOp< A,A >,
                     MultOp< B,B >
                    >
             >,
       V >
```

Building the Class

In C++ expression is the following calls

```
operator>=( sqrt( operator-( operator*(a,a),
                             operator*(b,b) ),
            v )
```

Expression class is built using the following operators

operators do not do the operation
operators return a class that can do the operation

```
template<class T,   class S>
        MultOp<T,S> operator*  (const T&, const S&);

template<class T,   class S>
        SubOp<T,S>  operator-  (const T&, const S&);

template<class T>
        SqrtOp<T>   sqrt       (const T&);

template<class T,   class S>
        GtEqOp<T,S> operator>= (const T&, const S&);
```

Doing the Work

Assignment operation of container class does the work

```
class Vector {
…
    template <class Node>
    void operator=(const Node& iN)
    {
       for(int i = 0 ;  i < size ;  ++i )
       {
          *this(i) = iN(i) ;
       }
    }
```

Compiler optimizes to original expression

```
for(int i = 0 ;  i < size ;  ++i)
 {
   *this(i) = sqrt( a(i)*a(i) -
                    b(i)*b(i)   ) >= v(i)
 }
```

Parts of Expression

```
SimpleSelector<Decay> d0Sel = abs(vMass-kD0Mass) < 100*k_MeV
```

Variables

define what methods to be accessed from the Candidate (e.g., Decay object)

```
vMass
```

Mathematical operators

transformation to apply to value obtained from Variables

```
abs( vMass - kD0Mass )
```

Comparison operators

constructs the class that can perform the comparison

```
<
```

Expression Variables

Purpose: holds functor to call when the expression is evaluated

```
template<class F> struct Var {
   Var(const F& iF = F() ) : m_f(iF) {}
   typedef F func_type;
   F m_f;
};
```

Example: Call Decay::mass()

Using generic std function classes

```
Var<mem_fun_ref_t<double, Decay> >
         vMass( mem_fun_ref( &Decay::mass ) );
```

Using a specialized helper class

```
Var<mass> vMass;
```

Mathematical Operators

Purpose: Transform the value of Variables

Implementation: define 12 operators plus math functions

4 methods(+,-,*,/) taking different arguments ({Var,Var}, {double,Var}, {Var, double})

Example: operator+ taking Var and double

```
typedef bind2nd<plus<double> > bind2plus;          calculates argument + stored_value
template <class F>
Var< Composite< F, bind2plus> >
operator+( const Var<F>& iVar, double iValue ) {
  typedef Composite<F, bind2plus> CompT;     Composite calculates func2( func1(argument ) );
  CompT temp( iVar.m_func, bind2nd( plus<double>(), iValue ));
  return Var<CompT>( temp );
}
```

returned object calculates `iVar.m_func( object ) + iValue`

Comparison Operators

Purpose: Construct the selection class

Example:

```
template<class T>
VarMethod<T::var_type, T::func_type, double, less<double> >
operator<(const T& iVar,  double iValue)
{  return VarMethod<...>(iVar.m_func, iValue); }
```

where

```
template< class T,        the type of object being compared (e.g. Decay)
          class F,        the mathematical transformation to apply to the object of type T
          class V,        the type of the value being compare with
          class TComp>    the comparison operation being applied (e.g. >)
struct VarMethod {
   VarMethod(F iF,  V iV ) : m_value(iV), m_func(iF) {}
   bool operator()(const T& iArg) {
      return m_comparison( m_func(iArg), m_value) ;
   }
   V m_value;   F m_func;   TComp m_comparison;
};
```

calculates `iVar.m_func(iArg) < iValue`

Compilation Equivalent

Compiler optimizations can turn

```
sel = abs( vMass   - kD0Mass    ) < 100*k_MeV &&
      abs( vEnergy - beamEnergy ) < 100*k_MeV;
```

Into code equivalent to

```
struct Temp {
  Temp( double iValue ) : m_beamEnergy(iValue) {}

  bool operator()(Decay& iDecay ) {
    return abs( iDecay.mass()   - kD0Mass      ) < 100*k_MeV
           &&
           abs( iDecay.energy() - m_beamEnergy ) < 100*k_MeV;
  }

  double m_beamEnergy;
};

sel = Temp( beamEnergy ) ;
```

Full Example

```
Result D0Filter::event(Frame& iFrame ) {
  Table<Track> tracks;
  extract(iFrame.record(kEvent), tracks);

  Item<BeamEnergy> beamEnergy;
  extract(iFrame.record(kRun), beamEnergy);

  ChargedPionList pions;    ChargedKaonList kaons;
  pions = tracks;           kaons = tracks;

  Var<mass> vMass;          Var<energy> vEnergy;
  SimpleSelector<Decay> sel = abs(vMass - kD0Mass) < 100*k_MeV
                     && abs(vEnergy - beamEnergy) < 100*k_MeV;

  DecayList d0List(sel);
  d0List  = kaons.minus() * pions.plus();
  d0List += kaons.minus() * pions.plus() *
            pions.minus() * pions.plus();

  return  d0List.size() ? kPass : kFailed;
}
```

Conclusion

C++ does not have to be a burden to physicists

Templates, operator overloading and exceptions can substantially reduce the work of getting an analysis done

Our experience is physicists will embrace libraries using these advanced concepts if they makes their jobs easier

even if they must initially learn new coding conventions